Feuille de révisions n°5 - Programmation en C : manipulation de pointeurs

1 Pointeurs et mémoire

R. 5-1 Décrire précisément l'évolution de la mémoire (tas et pile) au cours de l'exécution des deux programmes ci-dessous, en déduire quel affichage qu'ils produiront.

```c
int a, b;
int* aa1 = &a;
int* bb1 = &b;

a = b;
b = b + 2;
int* aa2 = &a;
int* bb2 = &b;

printf("a==b : %d\n",a==b);
printf("aa1==bb1 : %d\n",aa1==bb1);
printf("aa1==aa2 : %d\n",aa1==aa2);
printf("bb1==bb2 : %d\n",bb1==bb2);
```

```c
char* s = "abc";
char** pp1 = &s;
char* p1 = s;
char c1 = s[0];
char* ac1 = &c1;

s = "baba";
char** pp2 = &s;
char* p2 = s;
char c2 = s[1];
char* ac2 = &c2;

printf("p1==p2 : %d\n",p1==p2);
printf("pp1==pp2 : %d\n",pp1==pp2);
printf("c1==c2 : %d\n",c1==c2);
printf("ac1==ac2 : %d\n",ac1==ac2);
```

Solution

affichages obtenus :

```
a == b : 0
aa1 == bb1 : 0
aa1 == aa2 : 1
bb1 == bb2 : 1
```

```
p1 == p2 : 0
pp1 == pp2 : 1
c1 == c2 : 1
ac1 == ac2 : 0
```

À retenir Lorsqu'on modifie une variable on change sa valeur, pas l'endroit où elle est enregistrée de sorte que son adresse reste inchangée.

R. 5-2 On suppose que les déclarations ci-contre sont faites dans le `main` d'un programme. Comment sont placées ces variables dans la pile ? En particulier, si l'écart entre les adresses de `a1` et `a2` correspond à 4 octets, quel écart y a-t-il entre les adresses des autres variables ?
Les adresses de `tab1[0]` et `tab2[0]` sont-elles comparables ?
Résumer le comportement lors des différentes déclarations de tableaux.

```c
int a1;
int a2;
int tab1[3];
int a3;
int* tab2=malloc(sizeof(int)*3);
int a4;
int* tab3;
int a5;
```

Solution

On donne ci-contre les adresses affichées (avec `%d`) pour chaque variable lors d'une exécution. Les valeurs particulières ne sont pas pertinentes, mais les écarts entre les adresses sont informatifs.

- Un `int` occupe 4 octets, donc l'écart entre `a1` et `a2` est de 4.
- Trois `int` occupent 12 octets, donc l'écart entre `a2` et `tab1` est de 12.
- Un `int` occupe 4 octets, donc l'écart entre `tab1[0]` et `a3` est de 4.
- Un pointeur occupe 8 octets, donc l'écart entre `a3` et `tab2` est de 8.
- Un `int` occupe 4 octets, donc l'écart entre `tab2` et `a4` est de 4.
- Un pointeur occupe 8 octets, donc l'écart entre `a4` et `tab3` est de 8.
- Un `int` occupe 4 octets, donc l'écart entre `tab3` et `a5` est de 4.

```
a1 : 1210010220
a2 : 1210010216
a3 : 1210010200
a4 : 1210010188
a5 : 1210010172

tab1[0]:1210010204
tab1[1]:1210010208
tab1[2]:1210010212
tab2:1210010192
tab3:1210010176
```

Les adresses de `tab1[0]` et `tab2[0]` ne sont pas comparables. L'adresse de `tab1[0]` est une adresse dans la pile car `tab1` est un tableau statique, tandis que celle de `tab2[0]` est une adresse du tas, car ce tableau a été alloué dynamiquement. En particulier l'adresse de `tab2[0]` n'est pas corellée à l'adresse de `tab2`.

```
tab1[0]:1210010204
tab2[0]:857621088
```

À retenir Les variables sont placées dans la pile dans l'ordre où elles sont déclarées. Les variables de type `int` occupent 4 octets. Les tableaux **statiques**, *i.e.* déclarés avec un taille littérale, sont stockés dans la pile. Ainsi déclarer un tableau statique de type `t` et de taille n est équivalent à n déclarations de variables de type `t`. Les pointeurs (vers n'importe quoi) occupent 8 octets, ainsi les tableaux (non statiques) occupent 8 octets dans la pile peu importe leur taille. Cette taille est fixée lorsque l'instruction `malloc` esr exécutée, on parle d'allocation de mémoire **dynamique**, et n'impacte que l'espace alors réservé dans le tas.

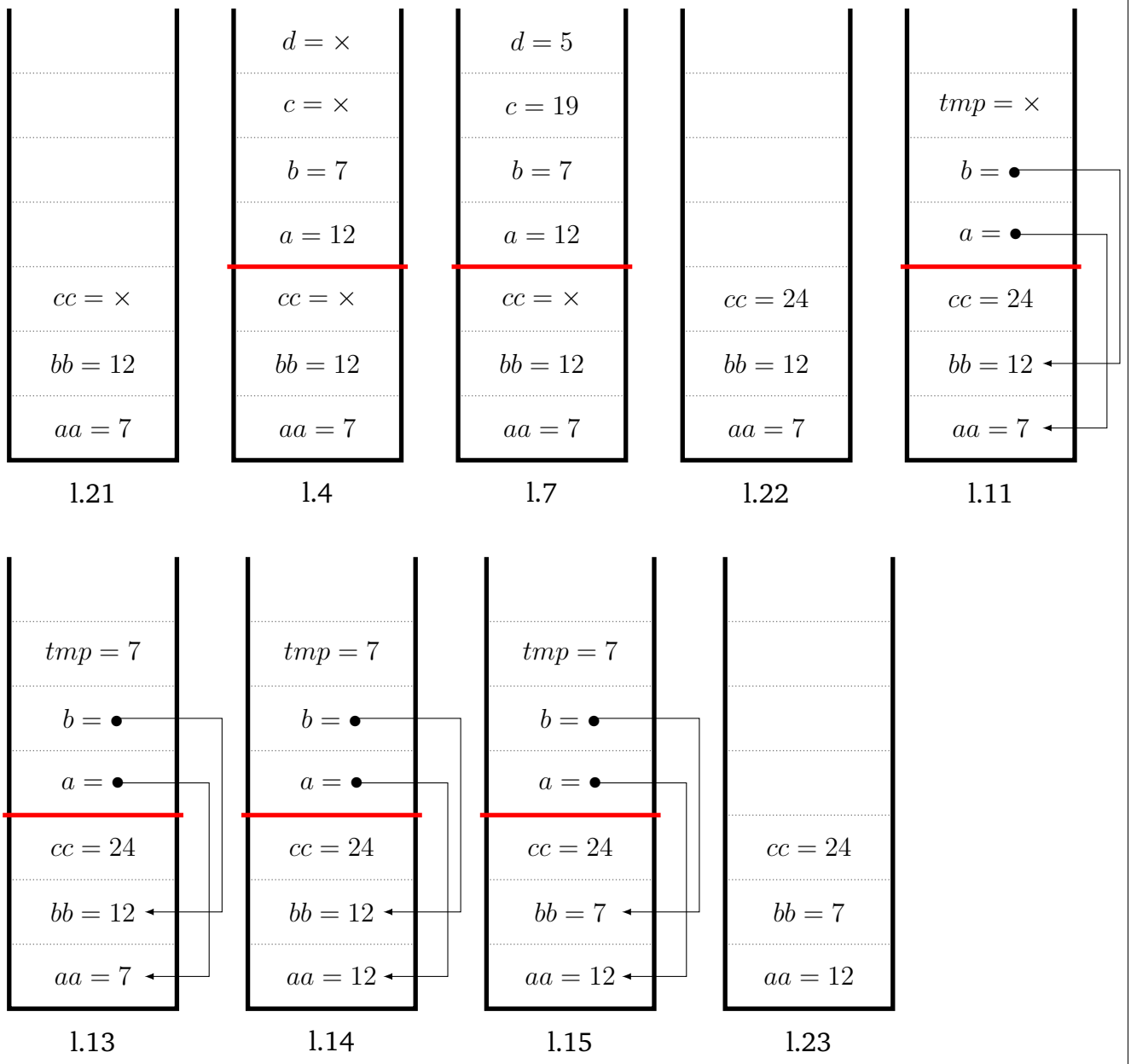
R. 5-3 Décrire l'évolution de la pile lors de l'exécution du programme ci-dessous.

```c
int f(int a, int b){
  // retourne ...
  int c = a + b;
  int d = a - b;
  return c + d;
}
```

```c
int g(int* a, int*b){
  // retourne ...
  int tmp = *a;
  *a = *b;
  *b = tmp;
  return tmp;
}
```

```c
int main(){
  int aa = 7;
  int bb = 12;
  int cc = f(bb,aa);
  cc = g(&aa,&bb);
  return 0;
}
```

Solution

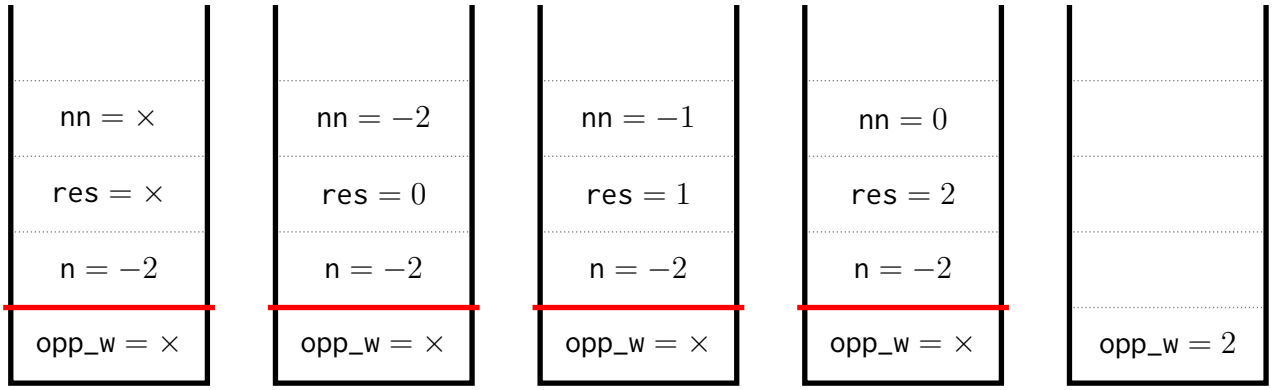

cc = ×
bb = 12
aa = 7
l.21
d = ×
c = ×
b = 7
a = 12
cc = ×
bb = 12
aa = 7
l.4
d = 5
c = 19
b = 7
a = 12
cc = ×
bb = 12
aa = 7
l.7
cc = 24
bb = 12
aa = 7
l.22
tmp = ×
b =
a =
cc = 24
bb = 12
aa = 7
l.11
tmp = 7
b =
a =
cc = 24
bb = 12
aa = 7
l.13
tmp = 7
b =
a =
cc = 24
bb = 12
aa = 12
l.14
tmp = 7
b =
a =
cc = 24
bb = 7
aa = 12
l.15
cc = 24
bb = 7
aa = 12
l.23

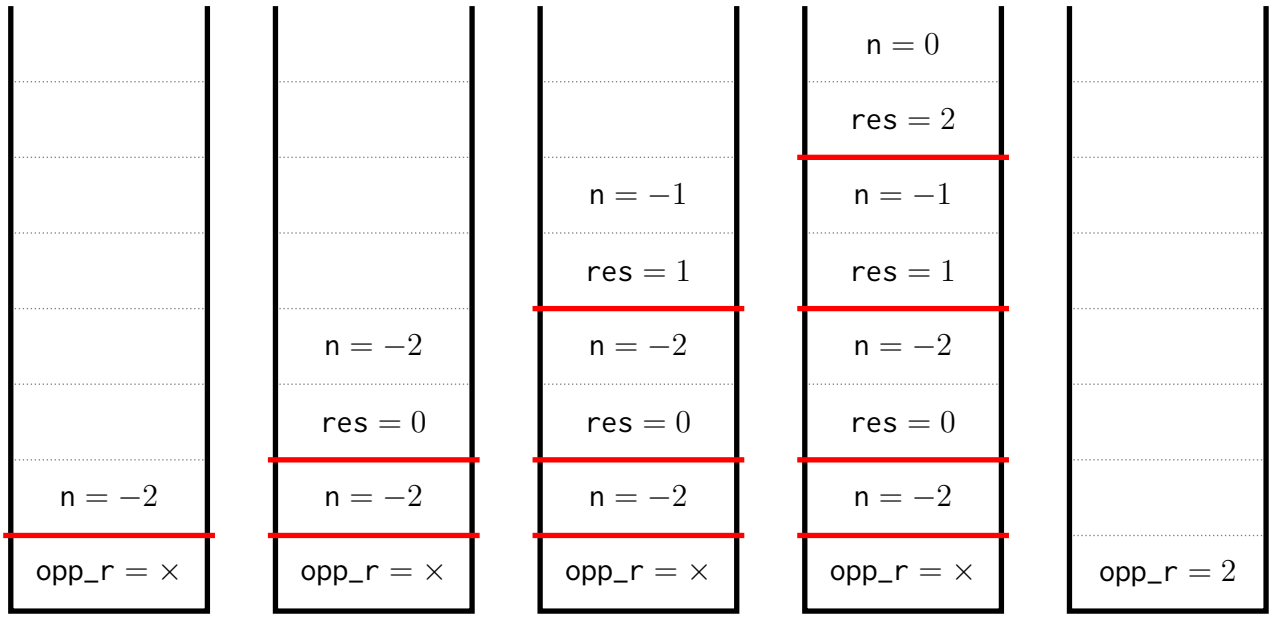
R. 5-4 On considère le code ci-après qui définit deux fonctions jumelles : l'une codée avec un `while` et l'autre est son analogue avec une fonction auxiliaire récursive. Décrire l'évolution de la pile lors de l'exécution de l'affectation `int opp_w = oppose_while(-2)`. Même question pour l'affectation `int opp_r = oppose_rec(-2)`. Synthétiser la différence de comportement entre les boucles `while` et les fonctions récursives. Le même comportement est-il observé en OCaml ?

```
1  int oppose_while(int n){
2    int res = 0;
3    int nn = n;
4    while(nn > 0){
5      nn = nn - 1;
6      res = res - 1;
7    }
8    while(nn < 0){
9      nn = nn + 1;
10     res = res + 1;
11   }
12   return res;
13 }
```

```
1  int aux (int n, int res){
2    if (n == 0){ return res;}
3    else{
4      if(n > 0){
5        return aux(n-1, res-1);
6      }
7      else{
8        return aux(n+1, res+1);
9      }
10   }
11 }
12
13 int oppose_rec(int n){
14   return aux(n,0);
15 }
```

Solution





l.13 | l.14 = l.1.1 | l.8 = l.1.2 | l.8 = l.1.3 | déf. de opp_r

À retenir Lorsqu'une fonction est appelée, dès le début est reservé sur la pile l'espac nécessaire pour stocker chacun des arguments et cahcune des variables locales à la fonction. Ainsi, à chaque appel récursif de l'espace est reservé sur la pile pour copier les valeurs des arguments dans ds nouvelles cases, tandis qu'avec une boucle while, les nouvelles valeurs des variables à chaque tour de boucle viennent écraser les anciennes valeurs. Ainsi, en C, on peut faire déborder la pile si on lance trop d'appels récursifs imbriqués. Le même problème peut survenir en OCaml avec des foncions récursives non récursives terminales.

R. 5-5 Pour les deux définitions de type suivantes (poly1 et poly2), définir une fonction qui crée une occurrence du polynôme $aX^2 + bX + c$ à partir des coefficients a, b et c.

```c
struct poly_s {
  int n;
  float* tab; //de taille n
};
typedef struct poly_s poly1;
typedef struct poly_s* poly2;
```

Solution

```c
poly1 cree_poly_deg2(float a, float b, float c){
  //cree une occurence du polynome aX^2+bX+c
  poly1 res;
  res.n = 3;
  res.tab = (float*) malloc (sizeof(float)*3);
  res.tab[0] = c;
  res.tab[1] = b;
  res.tab[2] = a;
  return res;
}
```

```c
poly2 new_poly_deg2 (float a, float b, float c){
  //cree une occurence du polynome aX^2+bX+c
```

```
  poly2 res = malloc (sizeof(struct poly_s));
  res->n = 3;
  res->tab = (float*) malloc (sizeof(float)*3);
  res->tab[0] = c;
  res->tab[1] = b;
  res->tab[2] = a;
  return res;
}
```

À retenir L'espace occupé par un objet de type `t*` est le même quel que soit le type `t`, c'est l'espace qu'occupe un pointeur. Pour remplir l'objet de type `t` au bout d'un tel pointeur, il faut avoir réservé l'espace pour un objet de type `t` (par déclaration sur la pile ou par `malloc` dans le tas selon les cas).

À retenir Pour retourner directement[a] une struct, on peut déclarer un objet de type cette struct, le remplir et le retourner. En revanche si on veut retourner un pointeur vers une struct qu'on a créée dans le corps de la fonction, celle-ci doit avoir été enregistrée dans le tas et non dans la pile, sans quoi elle est écrasée lorsqu'on sort de la fonction. Il faut donc faire un `malloc`.

a. i.e. pas à travers un pointeur

R. 5-6 On considère le code suivant. Décrire l'évolution de la mémoire au cours de son exécution, en déduire le comportement observé (en particulier l'affichage). Même question si on on remplace la ligne 3 par la ligne 4 d'une part, et si on décommente la ligne 16 d'autre part.

```
1  char* c1 = "mpsi";
2  char* c2 = "mpsi";
3  char c3[5]="mpsi";
4  //char c3[4]="mpsi";
5  char c4[5];
6  strcpy(c4,c1);//de string.h
7  char* c5 = (char*)
   ↪  malloc(5*sizeof(char));
8  strcpy(c5,c1);//de string.h
9  printf("Adresses enregistrées par\n");
10 printf("c1 : %p / c2: %p\n",c1,c2);
11 printf("c3 : %p / c4: %p\n",c3,c4);
12 printf("c5 : %p\n\n",c5);
13
14 printf("Chaîne pointées par\n");
15 printf("c1:%s/c2:%s\n",c1,c2);
16 //c2[2]='2';
17 printf("c3:%s/c4:%s/c5:%s\n",c3,c4,c5);
18 c3[2]='2';
19 printf("c3:%s/c4:%s/c5:%s\n",c3,c4,c5);
```

Solution

On obtient par exemple l'affichage ci-contre.

```
1 Adresses enregistrées par
2 c1 : 0x557aafecf004 / c2: 0x557aafecf004
3 c3 : 0x7fff115a7753 / c4: 0x7fff115a774e
4 c5 : 0x557ab068b260
5
6 Chaîne pointées par
7 c1:mpsi/c2:mpsi
8 c3:mpsi/c4:mpsi/c5:mpsi
9 c3:mp2i/c4:mpsi/c5:mpsi
```

→ Les chaînes explicitement déclarées aux lignes 1 et 2 sont stockées dès la compilation

dans un même endroit de la mémoire qu'on ne peut pas modifier. On le constate car les valeurs de `c1` et `c2`, les adresses qu'elles enregistrent sont les mêmes.

→ Les chaînes `c3` et `c4`, si elles ne sont pas remplies de la même manière, sont déclarées de la même manière : comme un tableau statique, un tableau dont la taille est une constante littérale, et sont donc enregistrées dans la pile. On peut s'en convaincre en ajoutant la déclaration d'un ou plusieurs entiers et en affichant leurs adresses. On voit que les adresses se ressemblent ce qui confirme l'emplacement sur la pile. En observant les écarts entre les adresses, on pourra remarquer que déclarer un objet `char c[5]` revient à déclarer cinq variables de type `char` qui sont alors empilées consécutivement sur la pile.

→ La chaîne `c5` est quant à elle enregistrée dans le tas.

Si on remplace la ligne 3 par la ligne 4, *i.e.* si on oublie de réserver la place pour le caractère de fin, l'affichage de `c5` termine par des caractères non maîtrisés et non désirés.
Si on décommente la ligne 16, l'exécution de cette ligne déclenche une erreur de segmentation. En effet la chaîne `c1` a été réservée dans une zone mémoire protégée, on ne peut pas la modifier (et heureusement sinon `c2` serait modifiée par la même occasion). Cette zone est protégée même s'il y a un seul pointeur vers cette chaîne, par exemple ici même si on supprime `c2`.

À retenir Les objets de type `char*` définis par une chaîne explicite sont gérés de manière particulière à la compilation. Lorsque plusieurs occurrences de la même chaîne sont déclarées, la chaîne est enregistrée une seule fois et on obtient alors plusieurs pointeurs vers la même zone mémoire. Les pointeurs obtenus peuvent être modifiés (pour pointer vers autre chose) mais la zone mémoire en question ne peut être modifiée (afin de ne pas dégrader la chaîne vers laquelle pointe peut-être un autre pointeur).

À retenir Pour une chaîne de n caractères il faut un tableau de $n+1$ `char` pour enregistrer aussi le caractère de fin de chaîne `'\0'`.